Exhibit E

'520 Patent	Infringed By

	String message = t.getMessage();
	if (message == null) {
	message = t.getClass().getCanonicalName();
	}
	message = String.format(Messages.Dalvik_Error_s, message);
	
	throw new DexException(message, t);
	}
	}

simulating execution of the byte codes of the clinit method against a memory without executing the byte codes to identify the static initialization of the array by the preloader;	The dx tool steps through the Java .class files, simulating execution of the bytecodes against a memory without executing the byte codes to identify the static initialization of the array by the preloader. For example, Android does not run Java .class files directly; instead, Java .class files are identified and translated into .dex files using the dx utility, where the dx utility simulates the effects of executing the bytecode in order to perform the translation. *See, e.g.*, dalvik\dx\src\com\android\dx\dex\cf\CfTranslator.java, and dalvik/dx/src/com/android/dx/cf/code/Simulator.java. These processes identify the static initialization of the array by simulating execution thereof. The Dalvik Video further describes source .class files (slides 41 and 42) for initialization, which includes converting/translating to .dex files (slide 44) and adding elements to a static array to initialize the data (slide 45). *See, e.g.*, time 29:50-32:00 and Dalvik Presentation, slides 41-45. *See e.g.*, dalvik/dx/src/com/android/dx/cf/code/Simulator.java, which explicitly provides a "Class which knows how to *simulate* the effects of executing bytecode." (Emphasis added.): /** * Class which knows how to simulate the effects of executing bytecode. * * Note: This class is not thread-safe. If multiple threads * need to use a single instance, they must synchronize access explicitly * between themselves. */ public class Simulator { /** * {@code non-null;} canned error message for local variable

’520 Patent	Infringed By

```
 * table mismatches
 */
private static final String LOCAL_MISMATCH_ERROR =
    “This is symptomatic of .class transformation tools that ignore “ +
    “local variable information.”;

/** {@code non-null;} machine to use when simulating */
private final Machine machine;

/** {@code non-null;} array of bytecode */
private final BytecodeArray code;

/** {@code non-null;} local variable information */
private final LocalVariableList localVariables;

/** {@code non-null;} visitor instance to use */
private final SimVisitor visitor;

/**
 * Constructs an instance.
 *
 * @param machine {@code non-null;} machine to use when simulating
 * @param method {@code non-null;} method data to use
 */
public Simulator(Machine machine, ConcreteMethod method) {
    if (machine == null) {
        throw new NullPointerException(“machine == null”);
    }

    if (method == null) {
        throw new NullPointerException(“method == null”);
    }

    this.machine = machine;
```

’520 Patent	Infringed By

```
        this.code = method.getCode();
        this.localVariables = method.getLocalVariables();
        this.visitor = new SimVisitor();
    }

    /**
     * Simulates the effect of executing the given basic block. This modifies
     * the passed-in frame to represent the end result.
     *
     * @param bb {@code non-null;} the basic block
     * @param frame {@code non-null;} frame to operate on
     */
    public void simulate(ByteBlock bb, Frame frame) {
        int end = bb.getEnd();

        visitor.setFrame(frame);

        try {
            for (int off = bb.getStart(); off < end; /*off*/) {
                int length = code.parseInstruction(off, visitor);
                visitor.setPreviousOffset(off);
                off += length;
            }
        } catch (SimException ex) {
            frame.annotate(ex);
            throw ex;
        }
    }

    /**
     * Simulates the effect of the instruction at the given offset, by
     * making appropriate calls on the given frame.
     *
     * @param offset {@code >= 0;} offset of the instruction to simulate
```

’520 Patent	Infringed By

```
 * @param frame {@code non-null;} frame to operate on
 * @return the length of the instruction, in bytes
 */
public int simulate(int offset, Frame frame) {
    visitor.setFrame(frame);
    return code.parseInstruction(offset, visitor);
}

/**
 * Constructs an “illegal top-of-stack” exception, for the stack
 * manipulation opcodes.
 */
private static SimException illegalTos() {
    return new SimException(“stack mismatch: illegal “ +
            “top-of-stack for opcode”);
}

/**
 * Bytecode visitor used during simulation.
 */
private class SimVisitor implements BytecodeArray.Visitor {
    /**
     * {@code non-null;} machine instance to use (just to avoid excessive
     * cross-object field access)
     */
    private final Machine machine;

    /**
     * {@code null-ok;} frame to use; set with each call to
     * {@link Simulator#simulate}
     */
    private Frame frame;

    /** offset of the previous bytecode */
```

’520 Patent	Infringed By

```
        private int previousOffset;

        /**
         * Constructs an instance.
         */
        public SimVisitor() {
            this.machine = Simulator.this.machine;
            this.frame = null;
        }
```

See, e.g., dalvik/dx/src/com/android/dx/cf/code/BytecodeArray.java:

```
  /**
   * Helper to deal with {@code newarray}.
   *
   * @param offset the offset to the {@code newarray} opcode itself
   * @param visitor {@code non-null;} visitor to use
   * @return instruction length, in bytes
   */
  private int parseNewarray(int offset, Visitor visitor) {
    int value = bytes.getUnsignedByte(offset + 1);
    CstType type;
    switch (value) {
      case ByteOps.NEWARRAY_BOOLEAN: {
        type = CstType.BOOLEAN_ARRAY;
        break;
      }
      case ByteOps.NEWARRAY_CHAR: {
        type = CstType.CHAR_ARRAY;
        break;
      }
      case ByteOps.NEWARRAY_DOUBLE: {
        type = CstType.DOUBLE_ARRAY;
        break;
```

’520 Patent	Infringed By

```
        }
        case ByteOps.NEWARRAY_FLOAT: {
            type = CstType.FLOAT_ARRAY;
            break;
        }
        case ByteOps.NEWARRAY_BYTE: {
            type = CstType.BYTE_ARRAY;
            break;
        }
        case ByteOps.NEWARRAY_SHORT: {
            type = CstType.SHORT_ARRAY;
            break;
        }
        case ByteOps.NEWARRAY_INT: {
            type = CstType.INT_ARRAY;
            break;
        }
        case ByteOps.NEWARRAY_LONG: {
            type = CstType.LONG_ARRAY;
            break;
        }
        default: {
            throw new SimException(“bad newarray code ” +
                    Hex.u1(value));
        }
    }

    // Revisit the previous bytecode to find out the length of the array
    int previousOffset = visitor.getPreviousOffset();
    ConstantParserVisitor constantVisitor = new ConstantParserVisitor();
    int arrayLength = 0;

    /*
     * For visitors that don’t record the previous offset, -1 will be
```

’520 Patent	Infringed By

```
 * seen here
 */
if (previousOffset >= 0) {
   parseInstruction(previousOffset, constantVisitor);
   if (constantVisitor.cst instanceof CstInteger &&
         constantVisitor.length + previousOffset == offset) {
      arrayLength = constantVisitor.value;

   }
}

/*
 * Try to match the array initialization idiom. For example, if the
 * subsequent code is initializing an int array, we are expecting the
 * following pattern repeatedly:
 *  dup
 *  push index
 *  push value
 *  *astore
 *
 * where the index value will be incrimented sequentially from 0 up.
 */
int nInit = 0;
int curOffset = offset+2;
int lastOffset = curOffset;
ArrayList<Constant> initVals = new ArrayList<Constant>();

if (arrayLength != 0) {
   while (true) {
      boolean punt = false;

      // First check if the next bytecode is dup
      int nextByte = bytes.getUnsignedByte(curOffset++);
      if (nextByte != ByteOps.DUP)
```

’520 Patent	Infringed By

```
    break;

// Next check if the expected array index is pushed to the stack
parseInstruction(curOffset, constantVisitor);
if (constantVisitor.length == 0 ||
        !(constantVisitor.cst instanceof CstInteger) ||
        constantVisitor.value != nInit)
    break;

// Next, fetch the init value and record it
curOffset += constantVisitor.length;

// Next find out what kind of constant is pushed onto the stack
parseInstruction(curOffset, constantVisitor);
if (constantVisitor.length == 0 ||
        !(constantVisitor.cst instanceof CstLiteralBits))
    break;

curOffset += constantVisitor.length;
initVals.add(constantVisitor.cst);

nextByte = bytes.getUnsignedByte(curOffset++);
// Now, check if the value is stored to the array properly
switch (value) {
    case ByteOps.NEWARRAY_BYTE:
    case ByteOps.NEWARRAY_BOOLEAN: {
        if (nextByte != ByteOps.BASTORE) {
            punt = true;
        }
        break;
    }
    case ByteOps.NEWARRAY_CHAR: {
        if (nextByte != ByteOps.CASTORE) {
            punt = true;
```

’520 Patent	Infringed By

	}
	break;
	}
	case ByteOps.NEWARRAY_DOUBLE: {
	if (nextByte != ByteOps.DASTORE) {
	punt = true;
	}
	break;
	}
	case ByteOps.NEWARRAY_FLOAT: {
	if (nextByte != ByteOps.FASTORE) {
	punt = true;
	}
	break;
	}
	case ByteOps.NEWARRAY_SHORT: {
	if (nextByte != ByteOps.SASTORE) {
	punt = true;
	}
	break;
	}
	case ByteOps.NEWARRAY_INT: {
	if (nextByte != ByteOps.IASTORE) {
	punt = true;
	}
	break;
	}
	case ByteOps.NEWARRAY_LONG: {
	if (nextByte != ByteOps.LASTORE) {
	punt = true;
	}
	break;
	}
	default:

'520 Patent	Infringed By

	punt = true;
	break;
	}
	if (punt) {
	break;
	}
	lastOffset = curOffset;
	nInit++;
	}
	}
	
	/*
	* For singleton arrays it is still more economical to
	* generate the aput.
	*/
	if (nInit < 2 || nInit != arrayLength) {
	visitor.visitNewarray(offset, 2, type, null);
	return 2;
	} else {
	visitor.visitNewarray(offset, lastOffset - offset, type, initVals);
	return lastOffset - offset;
	}
	}
	See also:
	dalvik/dx/src/com/android/dx/cf/code/RopperMachine.java; and
	dalvik/vm/interp/Interp.c.

storing into an output file an instruction requesting the static initialization of the array;	The dx tool stores an instruction requesting the static initialization of the array. This process is described, for example, in the Dalvik Video at time 29:50-32:00 and Dalvik Presentation slides 41-45. *See, e.g.*, Main.java in the dexer package \dalvik\dx\src\com\android\dx\command\dexer: /** * Converts {@link #outputDex} into a {@code byte[]}, write

’520 Patent	Infringed By

and	* it out to the proper file (if any), and also do whatever human-oriented
	* dumping is required.
	*
	* @return {@code null-ok;} the converted {@code byte[]} or {@code null}
	* if there was a problem
	*/
	private static byte[] writeDex() {
	byte[] outArray = null;
	
	try {
	OutputStream out = null;
	OutputStream humanOutRaw = null;
	OutputStreamWriter humanOut = null;
	try {
	if (args.humanOutName != null) {
	humanOutRaw = openOutput(args.humanOutName);
	humanOut = new OutputStreamWriter(humanOutRaw);
	}
	
	if (args.methodToDump != null) {
	/*
	* Simply dump the requested method. Note: The call
	* to toDex() is required just to get the underlying
	* structures ready.
	*/
	outputDex.toDex(null, false);
	dumpMethod(outputDex, args.methodToDump, humanOut);
	} else {
	/*
	* This is the usual case: Create an output .dex file,
	* and write it, dump it, etc.
	*/
	outArray = outputDex.toDex(humanOut, args.verboseDump);

’520 Patent	Infringed By

```
                        if ((args.outName != null) && !args.jarOutput) {
                            out = openOutput(args.outName);
                            out.write(outArray);
                        }
                    }

                    if (args.statistics) {
                        DxConsole.out.println(outputDex.getStatistics().toHuman());
                    }
                } finally {
                    if (humanOut != null) {
                        humanOut.flush();
                    }
                    closeOutput(out);
                    closeOutput(humanOutRaw);
                }
            } catch (Exception ex) {
                if (args.debug) {
                    DxConsole.err.println("\ntrouble writing output:");
                    ex.printStackTrace(DxConsole.err);
                } else {
                    DxConsole.err.println("\ntrouble writing output: " +
                                          ex.getMessage());
                }
                return null;
            }

            return outArray;
        }
```

See, e.g., RopperMachine.java:

```
                 * Java library array constructor.
                 */
                Type componentType = ((CstType) cst).getClassType();
```

’520 Patent	Infringed By

```
for (int i = 0; i < sourceCount; i++) {
      componentType = componentType.getComponentType();
}

RegisterSpec classReg =
      RegisterSpec.make(dest.getReg(), Type.CLASS);

if (componentType.isPrimitive()) {
      /*
       * The component type is primitive (e.g., int as opposed
       * to Integer), so we have to fetch the corresponding
       * TYPE class.
       */
      CstFieldRef typeField =
            CstFieldRef.forPrimitiveType(componentType);
      insn = new ThrowingCstInsn(Rops.GET_STATIC_OBJECT, pos,
                                 RegisterSpecList.EMPTY,
                                 catches, typeField);
} else {
      /*
       * The component type is an object type, so just make a
       * normal class reference.
       */
      insn = new ThrowingCstInsn(Rops.CONST_OBJECT, pos,
                                 RegisterSpecList.EMPTY, catches,
                                 new CstType(componentType));
}

insns.add(insn);

// Add a move-result-pseudo for the get-static or const
rop = Rops.opMoveResultPseudo(classReg.getType());
insn = new PlainInsn(rop, pos, classReg, RegisterSpecList.EMPTY);
insns.add(insn);
```

’520 Patent	Infringed By

```
                /*
                 * Add a call to the “multianewarray method,” that is,
                 * Array.newInstance(class, dims). Note: The result type
                 * of newInstance() is Object, which is why the last
                 * instruction in this sequence is a cast to the right
                 * type for the original instruction.
                 */

                RegisterSpec objectReg =
                        RegisterSpec.make(dest.getReg(), Type.OBJECT);

                insn = new ThrowingCstInsn(
                            Rops.opInvokeStatic(MULTIANEWARRAY_METHOD.getPrototype()),
                            pos, RegisterSpecList.make(classReg, dimsReg),
                            catches, MULTIANEWARRAY_METHOD);
                insns.add(insn);

                // Add a move-result.
                rop = Rops.opMoveResult(MULTIANEWARRAY_METHOD.getPrototype()
                            .getReturnType());
                insn = new PlainInsn(rop, pos, objectReg, RegisterSpecList.EMPTY);
                insns.add(insn);

                /*
                 * And finally, set up for the remainder of this method to
                 * add an appropriate cast.
                 */

                opcode = ByteOps.CHECKCAST;
                sources = RegisterSpecList.make(objectReg);
            } else if (opcode == ByteOps.JSR) {
                // JSR has no Rop instruction
                hasJsr = true;
```

’520 Patent	Infringed By

```
        return;
    } else if (opcode == ByteOps.RET) {
        try {
            returnAddress = (ReturnAddress)arg(0);
        } catch (ClassCastException ex) {
            throw new RuntimeException(
                    “Argument to RET was not a ReturnAddress”, ex);
        }
        // RET has no Rop instruction.
        return;
    }

    ropOpcode = jopToRopOpcode(opcode, cst);
    rop = Rops.ropFor(ropOpcode, destType, sources, cst);

    Insn moveResult = null;
    if (dest != null && rop.isCallLike()) {
        /*
         * We’re going to want to have a move-result in the next
         * basic block.
         */
        extraBlockCount++;

        moveResult = new PlainInsn(
                Rops.opMoveResult(((CstMethodRef) cst).getPrototype()
                .getReturnType()), pos, dest, RegisterSpecList.EMPTY);

        dest = null;
    } else if (dest != null && rop.canThrow()) {
        /*
         * We’re going to want to have a move-result-pseudo in the
         * next basic block.
         */
        extraBlockCount++;
```

’520 Patent	Infringed By

```
        moveResult = new PlainInsn(
                Rops.opMoveResultPseudo(dest.getTypeBearer()),
                pos, dest, RegisterSpecList.EMPTY);

        dest = null;
    }
    if (ropOpcode == RegOps.NEW_ARRAY) {
        /*
         * In the original bytecode, this was either a primitive
         * array constructor “newarray” or an object array
         * constructor “anewarray”. In the former case, there is
         * no explicit constant, and in the latter, the constant
         * is for the element type and not the array type. The rop
         * instruction form for both of these is supposed to be
         * the resulting array type, so we initialize / alter
         * “cst” here, accordingly. Conveniently enough, the rop
         * opcode already gets constructed with the proper array
         * type.
         */
        cst = CstType.intern(rop.getResult());
    } else if ((cst == null) && (sourceCount == 2)) {
        TypeBearer lastType = sources.get(1).getTypeBearer();

        if (lastType.isConstant()
                && advice.hasConstantOperation(rop,
                sources.get(0), sources.get(1))) {
            /*
             * The target architecture has an instruction that can
             * build in the constant found in the second argument,
             * so pull it out of the sources and just use it as a
             * constant here.
             */
            cst = (Constant) lastType;
```

’520 Patent	Infringed By

```
                sources = sources.withoutLast();
                rop = Rops.ropFor(ropOpcode, destType, sources, cst);
            }
        }

        SwitchList cases = getAuxCases();
        ArrayList<Constant> initValues = getInitValues();
        boolean canThrow = rop.canThrow();

        blockCanThrow |= canThrow;

        if (cases != null) {
            if (cases.size() == 0) {
                // It’s a default-only switch statement. It can happen!
                insn = new PlainInsn(Rops.GOTO, pos, null,
                                     RegisterSpecList.EMPTY);
                primarySuccessorIndex = 0;
            } else {
                IntList values = cases.getValues();
                insn = new SwitchInsn(rop, pos, dest, sources, values);
                primarySuccessorIndex = values.size();
            }
        } else if (ropOpcode == RegOps.RETURN) {
            /*
             * Returns get turned into the combination of a move (if
             * non-void and if the return doesn’t already mention
             * register 0) and a goto (to the return block).
             */
            if (sources.size() != 0) {
                RegisterSpec source = sources.get(0);
                TypeBearer type = source.getTypeBearer();
                if (source.getReg() != 0) {
                    insns.add(new PlainInsn(Rops.opMove(type), pos,
                                            RegisterSpec.make(0, type),
```

’520 Patent	Infringed By

```
                                                    source));
                    }
                }
                insn = new PlainInsn(Rops.GOTO, pos, null, RegisterSpecList.EMPTY);
                primarySuccessorIndex = 0;
                updateReturnOp(rop, pos);
                returns = true;
            } else if (cst != null) {
                if (canThrow) {
                    insn =
                            new ThrowingCstInsn(rop, pos, sources, catches, cst);
                    catchesUsed = true;
                    primarySuccessorIndex = catches.size();
                } else {
                    insn = new PlainCstInsn(rop, pos, dest, sources, cst);
                }
            } else if (canThrow) {
                insn = new ThrowingInsn(rop, pos, sources, catches);
                catchesUsed = true;
                if (opcode == ByteOps.ATHROW) {
                    /*
                     * The op athrow is the only one where it’s possible
                     * to have non-empty successors and yet not have a
                     * primary successor.
                     */
                    primarySuccessorIndex = -1;
                } else {
                    primarySuccessorIndex = catches.size();
                }
            } else {
                insn = new PlainInsn(rop, pos, dest, sources);
            }

            insns.add(insn);
```

’520 Patent	Infringed By

```
            if (moveResult != null) {
                insns.add(moveResult);
            }

            /*
             * If initValues is non-null, it means that the parser has
             * seen a group of compatible constant initialization
             * bytecodes that are applied to the current newarray. The
             * action we take here is to convert these initialization
             * bytecodes into a single fill-array-data ROP which lays out
             * all the constant values in a table.
             */
            if (initValues != null) {
                extraBlockCount++;
                insn = new FillArrayDataInsn(Rops.FILL_ARRAY_DATA, pos,
                        RegisterSpecList.make(moveResult.getResult()), initValues,
                        cst);
                insns.add(insn);
            }
        }
```

dalvik/dx/src/com/android/dx/cf/code/RopperMachine.java.

See also:
dalvik/dx/src/com/android/dx/cf/code/Simulator.java;
dalvik/dx/src/com/android/dx/cf/code/BytecodeArray.java;
dalvik/dx/src/com/android/dx/cf/code/RopperMachine.java; and
dalvik/vm/interp/Interp.c.

Classes.dex is a usual output of the dx tool. In the skeletonapp example, dx creates classes.dex, in which the instructions requesting the static initialization of the two arrays has been stored:

```
000400:                                        |[000400]
com.example.android.skeletonapp.skeletonapp.<clinit>:()V
```

'520 Patent	Infringed By
	000410: 1241 |0000: const/4 v1, #int 4 // #4 000412: 2310 1200 |0001: new-array v0, v1, [I // type@0012 000416: 2600 0d00 0000 |0003: fill-array-data v0, 00000010 // +0000000d 00041c: 6900 0700 |0006: sput-object v0, Lcom/example/android/skeletonapp/skeletonapp;.tester:[I // field@0007 000420: 2310 1200 |0008: new-array v0, v1, [I // type@0012 000424: 2600 1200 0000 |000a: fill-array-data v0, 0000001c // +00000012 00042a: 6900 0400 |000d: sput-object v0, Lcom/example/android/skeletonapp/skeletonapp;.bigger:[I // field@0004 00042e: 0e00 |000f: return-void 000430: 0003 0400 0400 0000 0100 0000 0200 ... |0010: array-data (12 units) 000448: 0003 0400 0400 0000 ff00 0000 7f00 ... |001c: array-data (12 units)

interpreting the instruction by a virtual machine to perform the static initialization of the array.	The Dalvik virtual machine interprets the instruction to perform the static initialization of the array. This process is described, for example, in the Dalvik Video at time 29:50-32:00 and Dalvik Presentation slides 41-45.
	
	See also:
	/*
	* Fill the array with predefined constant values.
	*
	* Returns true if job is completed, otherwise false to indicate that
	* an exception has been thrown.
	*/
	bool dvmInterpHandleFillArrayData(ArrayObject* arrayObj, const u2* arrayData)
	{
	u2 width;
	u4 size;
	
	if (arrayObj == NULL) {
	dvmThrowException("Ljava/lang/NullPointerException;", NULL);
	return false;
	}
	
	/*
	* Array data table format: